Specifying Concurrent Problems: Beyond Linearizability and up to Tasks

Armando Castañeda, Sergio Rajsbaum, Michel Raynal

Specifying Concurrent Problems: Beyond Linearizability and up to Tasks

(Extended Abstract)*

Armando Castañeda[1]**, Sergio Rajsbaum[1]***, and Michel Raynal[2]†

[1] Instituto de Matemáticas, UNAM, México D.F, 04510, México
armando.castaneda@im.unam.mx,rajsbaum@im.unam.mx
[2] IUF & IRISA (Université de Rennes), 35042 Rennes, France raynal@irisa.fr

Abstract. Tasks and objects are two predominant ways of specifying distributed problems. A *task* specifies for each set of processes (which may run concurrently) the valid outputs of the processes. An *object* specifies the outputs the object may produce when it is accessed sequentially. Each one requires its own *implementation* notion, to tell when an execution satisfies the specification. For objects *linearizability* is commonly used, while for tasks implementation notions are less explored.
Sequential specifications are very convenient, especially important is the *locality* property of linearizability, which states that linearizable objects compose for free into a linearizable object. However, most well-known tasks have no sequential specification. Also, tasks have no clear locality property.
The paper introduces the notion of *interval-sequential* object. The corresponding implementation notion of *interval-linearizability* generalizes linearizability. Interval-linearizability allows to specify any task. However, there are sequential one-shot objects that cannot be expressed as tasks, under the simplest interpretation of a task. The paper also shows that a natural extension of the notion of a task is expressive enough to specify any interval-sequential object.

Keywords: Concurrent object, task, linearizability, sequential specification.

1 Introduction

Concurrent objects Distributed computer scientists excel at thinking concurrently, and building large distributed programs that work under difficult conditions where processes experience asynchrony and failures. Yet, they evade thinking about *concurrent* problem specifications. A central paradigm is that of a

* Full version in http://arxiv.org/abs/1507.00073
** Partially supported by UNAM-PAPIIT
*** Partially supported by LAISLA-CONACYT and UNAM-PAPIIT
† Partially supported by the French ANR project DISPLEXITY, and the Franco-German ANR project DISCMAT

shared object that processes may access concurrently [19, 22], but the object is specified in terms of a sequential specification, i.e., an automaton describing the outputs the object produces only when it is accessed sequentially. Thus, a concurrent algorithm seeks to emulate an allowed sequential behavior.

There are various ways of defining what it means for an algorithm to *implement* an object, namely, that it satisfies its sequential specification. One of the most popular consistency conditions is *linearizability* [20]. An implementation is *linearizable* if each of its executions is *linearizable*: intuitively, for each operation call, it is possible to find a unique point in the interval of real-time defined by the invocation and response of the operation, and these *linearization points* induce a valid sequential execution. Linearizability is very popular to design components of large systems because it is *local*, namely, one can consider linearizable object implementations in isolation and *compose* them for free, without sacrificing linearizability of the whole system [11]. Also, linearizability is a *non-blocking* property, which means that a pending invocation (of a total operation) is never required to wait for another pending invocation to complete.

Linearizability has various desirable properties, additionally to being local and non-blocking: it allows talking about the state of an object, interactions among operations is captured by side-effects on object states; documentation size of an object is linear in the number of operations; new operations can be added without changing descriptions of old operations. However, as we argue here, linearizability is sometimes too restrictive. First, there are problems which have no sequential specifications (more on this below). Second, some problems are more naturally and succinctly defined in term of concurrent behaviors. Third, as it is well known, the specification of a problem should be as general as possible, to allow maximum flexibility to both programmers and program executions.

Distributed tasks Another predominant way of specifying a one-shot distributed problem, especially in distributed computability, is through the notion of a *task*. Several tasks have been intensively studied in distributed computability, leading to an understanding of their relative power [18], to the design of simulations between models [5], and to the development of a deep connection between distributed computing and topology [17]. Formally, a task is specified by an input/output relation, defining for each set of processes that may run concurrently, and each assignment of inputs to the processes in the set, the valid outputs of the processes. Implementation notions for tasks are less explored, and they are not as elegant as linearizability. In practice, task and implementation are usually described operationally, somewhat informally. One of the versions widely used is that an algorithm *implements* a task if, in every execution where a set of processes participate (run to completion, and the other crash from the beginning), input and outputs satisfy the task specification.

Tasks and objects model in a very different way the concurrency that naturally arises in distributed systems: while tasks explicitly state what might happen when a set of processes run concurrently, objects only specify what happens when processes access the object sequentially.

It is remarkable that these two approaches have largely remained independent, while the main distributed computing paradigm, *consensus*, is central to both. Neiger [21] noticed this and proposed a generalization of linearizability called *set-linearizability*. He discussed that there are tasks, like *immediate snapshot* [4], with no natural specification as sequential objects. An object modeling the immediate snapshot task is necessarily stronger than the immediate snapshot task, because such an object implements test-and-set. In contrast there are read/write algorithms solving the immediate snapshot task and it is well-known that there are no read/write linearizable implementations of test-and-set. Therefore, Neiger proposed the notion of a *set-sequential* object, that allows a set of processes to access an object simultaneously. Then, one can define an immediate snapshot set-sequential object, and there are *set-linearizable* implementations.

Contributions We propose the notion of an *interval-sequential* concurrent object, a framework in which an object is specified by an automaton that can express any concurrency pattern of overlapping invocations of operations, that might occur in an execution (although one is not forced to describe all of them). The automaton is a direct generalization of the automaton of a sequential object, except that transitions are labeled with sets of invocations and responses, allowing operations to span several consecutive transitions. The corresponding implementation notion of *interval-linearizability* generalizes linearizability and set-linearizability, and allows to associate states along the interval of execution of an operation. While linearizing an execution requires finding linearization *points*, in interval-linearizability one needs to identify a linearization *interval* for each operation (the intervals might overlap). Remarkably, this general notion remains local and non-blocking. We show that important tasks have no specification neither as a sequential object nor as a set-sequential object, but they can be naturally expressed as interval-sequential objects.

Establishing the relationship between tasks and (sequential, set-sequential and interval-sequential) automaton-based specifications is subtle, because tasks admit several natural interpretations. Interval-linearizability is a framework that allows to specify any task, however, there are sequential one-shot objects that cannot be expressed as tasks, under the simplest interpretation of what it means to solve a task. However, a natural extension of the notion of solving a task, which we call *refined tasks*, has the same expressive power to specify one-shot concurrent problems, hence strictly more than sequential and set-sequential objects. See Figure 1. Interval-linearizability goes beyond unifying sequentially specified objects and tasks, it sheds new light on both of them. On the one hand, interval-sequential linearizability provides an explicit operational semantics for a task (whose semantics, as we argue here, is not well understood), gives a more precise imple-

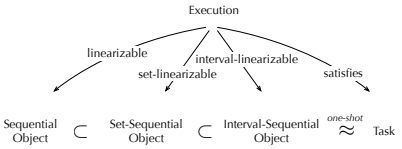


Fig. 1. Equivalence between *refined* tasks and one-shot interval-sequential objects.

mentation notion, and brings a locality property to tasks. On the other hand, tasks provide a static specification for automaton-based formalisms such as sequential, set-sequential and interval-sequential objects.

Finally, Shavit [24] summarizes beautifully the common knowledge state that "it is infinitely easier and more intuitive for us humans to specify how abstract data structures behave in a sequential setting." We hope interval-linearizability opens the possibility of facilitating reasoning about concurrent specifications, when no sequential specifications are possible.

All proofs and additional examples can be found in [7].

Related work Neiger proposed unifying sequential objects and tasks, using set-linearizability [21]. Later on, it was again observed that for some concurrent objects it is impossible to provide a sequential specification, and *concurrency-aware* linearizability was defined [16] (still, no locality properties were proved). Set linearizability and concurrency-aware linearizability are closely related and both are strictly less powerful than interval linearizability to model tasks. Transforming the question of wait-free read/write solvability of a one-shot sequential object, into the question of solvability of a task was suggested in [13]. The refined tasks we propose here is reminiscent to the construction in [13] Linearizability can be used in an operation that must wait for some other thread to establish a precondition, by defining two linearization points, representing a request and a follow-up [23]. These points are reminiscent of the interval used to define an interval-linearization. *Higher dimensional automata* are used to model execution of concurrent operations, and are the most expressive model among other common operations [14]. They can model transitions which consist of sets of operations, and hence are related to set-linearizability, but do not naturally model interval-linearizability. There is work on partial order semantics of programs, including more flexible notions of linearizability, relating two arbitrary sets of histories [10], although no compositionality result is proved, and concurrent executions are not explicitly studied. It is worth exploring this direction further, as the properties hold for concurrent executions, and it establishes that linearizability implies observational refinement, which usually entails compositionality (see, e.g., [15]).

2 Limitations of linearizability and set-linearizability

Sometimes we work with objects with two operations, but that are intended to be used as one. For instance, a snapshot object [1] has operations write() and snapshot(). This object has a sequential specification and there are linearizable read/write algorithms implementing it (see, e.g., [19, 22]). But many times, a snapshot object is used in a canonical way, namely, each time a process invokes write(), immediately after it always invokes snapshot(). Indeed, one would like to think of such an object as providing a single operation, write_snapshot(), invoked with a value x to be deposited in the object, and when the operation returns, it gives back to the invoking process a snapshot of the contents of the object. It

turns out that this write-snapshot object has neither a natural sequential nor a set-sequential specification. However, it can be specified as a task and actually is implementable from read/write registers.

As observed in [21], in a sequential specification of write-snapshot any of its executions can be seen as if all invocations occurred one after the other, in some order. Thus, always there is a first invocation, which must output the set containing only its input value, and hence could solve test-and-set, contradicting the fact that test-and-set cannot be implemented from read/write registers. Neiger noted this problem in the context of the immediate snapshot task. He proposed in [21] the idea that a specification should allow to express that sets of operations that can be concurrent. He called this notion *set-linearizability*. In set-linearizability, an execution accepted by a *set-sequential* automaton is a sequence of non-empty sets with operations, and each set denotes operations that are executed concurrently. While set-linearizability is sufficient to model the immediate-snapshot task, it is not enough for specifying the write-snapshot task, and most other tasks.

In set-linearizability, in the execution in Figure 2, one has to decide if the operation of q goes together with the one of p or r. In either case, in the resulting execution a process seems to predict a future operation. The problem is that there are operations that are affected by several operations that are not concurrent. This cannot be expressed as a set-sequential execution. Hence, to succinctly express this type of behavior, we need a more flexible framework in which it is possible to express that an operation happens in an interval of time that can be affected by several operations.

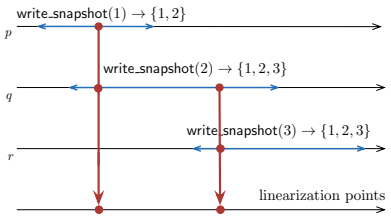


Fig. 2. A write-snapshot execution that is not set-linearizable.

To deal with these problematic tasks, one is tempted to separate an operation into two operations, set() and get(). The first communicates the input value of a process, while the second produces an output value to a process. For instance, k-set agreement is easily transformed into an object with a sequential specification, by accessing it through set() to deposit a value into the object and get() to obtain one of the values in the object. In fact, every task can be represented as a sequential object by splitting the operation of the task in two operations.

Separating an operation into a proposal operation and a returning operation has several problems (although it is useful in other contexts [23]). First, the program is forced to produce two operations, and wait for two responses. There is a consequent loss of clarity in the code of the program, in addition to a loss in performance, incurred by a two-round trip delay. Also, the intended meaning of linearization points is lost; an operation is now linearized at *two* linearization points. Furthermore, the resulting object may be more powerful; a phenomenon that has been observed several times in the context of iterated models e.g. [9].

Additional examples of problematic tasks Several tasks are problematic for dealing with them through linearizability, and have no deterministic sequential specifications. Some have been studied in the past, such as the following.

- *adopt-commit* [12] is useful to implement round-based protocols for set-agreement and consensus. Given an input u to the object, the result is an output of the form $(commit, v)$ or $(adopt, v)$, where *commit/adopt* is a decision that indicates whether the process should decide value v immediately or adopt it as its preferred value in later rounds of the protocol.
- *conflict detection* [3] has been shown to be equivalent to the adopt-commit. Roughly, if at least two different values are proposed concurrently at least one process outputs true.
- *safe-consensus* [2], a weakening of consensus, where the agreement condition of consensus is retained, but the validity condition becomes: if the first process to invoke it returns before any other process invokes it, then it outputs its input; otherwise the consensus output can be arbitrary, not even the input of any process.
- *immediate snapshot* [4], which plays an important role in distributed computability [17]. A process can write a value to the shared memory using this operation, and gets back a snapshot of the shared memory, such that the snapshot occurs immediately after the write.
- *k-set agreement* [8], where processes agree on at most k input values.
- *Exchanger* [16], is a Java object that serves as a synchronization point at which threads can pair up and atomically swap elements.

3 Concurrent Objects

3.1 System model

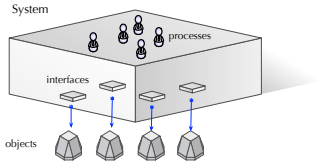
The presentation follows [6, 20, 22]. The *system* consists of n asynchronous sequential processes, $P = \{p_1, \ldots, p_n\}$, which communicate through a set of concurrent objects, OBS. Given a set OP of operations offered by the objects of the system to the processes P, let Inv be the set of all invocations to operations that can be issued by a process in a system, and Res be the set of all responses to the invocations in Inv. There are functions: (1) $id : Inv \to P$, (2) $Inv \to OP$, (3) $Res \to OP$, (4) $Res \to Inv$ and (5) $obj : OP \to OBS$, where $id(in)$ tells which process invoked $in \in Inv$, $op(in)$ tells which operation was invoked, $op(r)$ tells which operation was responded, $res(r)$ tells which invocation corresponds to $r \in Res$, and $obj(oper)$ indicates the object that offers operation $oper$. There is an induced function $id : Res \to P$ defined by $id(r) = id(res(r))$. Also, induced functions $obj : Inv \to OBS$ defined by $obj(in) = obj(op(in))$, and $obj : Res \to OBS$ defined by $obj(r) = obj(op(r))$. The set of operations of an object X, $OP(X)$, consists of all operations $oper$, with $obj(oper) = X$. Similarly, $Inv(X)$ and $Res(X)$ are resp. the set of invocations and responses of X.

A *process* is a deterministic automaton that interacts with the objects in OBS. It produces a sequence of steps, where a *step* is an invocation of an object's operation, or reacting to an object's response (including local processing).

Consider the set of all operations OP of objects in OBS, and all the corresponding possible invocations Inv and responses Res. A process p is an automaton (Σ, ν, τ), with states Σ and functions ν, τ that describe the interaction of the process with the objects. Often there is also a set of initial states $\Sigma_0 \subseteq \Sigma$. Intuitively, if p is in state σ and $\nu(\sigma) = (op, X)$ then in its next step p will apply operation op to object X. Based on its current state, X will return a response r to p and will enter a new state, in accordance to its transition relation. Finally, p will enter state $\tau(\sigma, r)$ as a result of the response it received from X.

A *system* consists of a set of processes, P, a set of objects OBS so that each $p \in P$ uses a subset of OBS, together with an initial state for each of the objects.

A *configuration* is a tuple consisting of the state of each process and each object, and a configuration is *initial* if each process and each object is in an initial state. An *execution* of the system is modeled by a sequence of events H arranged in a total order $\widehat{H} = (H, <_H)$, where each event is an invocation $in \in Inv$ or a response $r \in Res$, that can be produced following the process automata, interacting with the objects. Namely, an execution starts, given any initial configuration, by having any process invoke an operation, according to its transition relation. In general, once a configuration is reached, the next event can be a response from an object to an operation of a process or an invocation of an operation by a process whose last invocation has been responded. Thus, an execution is well-formed, in the sense that it consists of an interleaving of invocations and responses to operations, where a processes invokes an operation only when its last invocation has been responded.



3.2 The notion of an *Interval-sequential* object

To generalize the usual notion of a sequential object e.g. [6, 20] instead of considering sequences of invocations and responses, we consider sequences of *sets* of invocations and responses. An *invoking concurrency class* $C \subseteq 2^{Inv}$, is a non-empty subset of Inv such that C contains at most one invocation by the same process. A *responding concurrency class* C, $C \subseteq 2^{Res}$, is defined similarly.

Interval-sequential execution An *interval-sequential execution* h is an alternating sequence of invoking and responding concurrency classes, starting in an invoking class, $h = I_0, R_0, I_1, R_1, \ldots, I_m, R_m$, where the following conditions are satisfied

1. For each $I_i \in h$, any two invocations $in_1, in_2 \in I_i$ are by different processes, $id(in_1) \neq id(in_2)$. Similarly, for $R_i \in h$ if $r_1, r_2 \in R_i$ then $id(r_1) \neq id(r_2)$,
2. Let $r \in R_i$ for some $R_i \in h$. There is $in \in I_j$ for some $j \leq i$, such that $res(r) = in$ and there is no other in' with $id(in) = id(in')$ and $in' \in I_{j'}$, $j < j' \leq i$.

It follows that an execution h consists of matching invocations and responses, perhaps with some pending invocations with no response.

Interval-sequential object An *interval-sequential* object X is a (not necessarily finite) Mealy state machine $(Q, 2^{Inv(X)}, 2^{Res(X)}, \delta)$ whose output values R are responding concurrency classes R of X, $R \subseteq 2^{Res(X)}$, are determined both by its current state $s \in Q$ and the current input $I \in 2^{Inv(X)}$, where I is an invoking concurrency class of X. There is a set of *initial states* Q_0 of X, $Q_0 \subseteq Q$. The transition relation $\delta \subseteq Q \times 2^{inv(X)} \times 2^{Res(X)} \times Q$ specifies both, the output of the automaton and its next state. If X is in state q and it receives as input a set of invocations I, then, if $(R, q') \in \delta(q, I)$, the meaning is that X may return the non-empty set of responses R and move to state q'. We stress that always both I and R are non-empty sets.

Interval-sequential execution of an object Consider an initial state $q_0 \in Q_0$ of X and a sequence of inputs $I_0, I_1, \ldots I_m$. Then a sequence of outputs that X may produce is $R_0, R_1, \ldots R_m$, where $(R_i, q_{i+1}) \in \delta(q_i, I_i)$. Then the *interval-sequential execution of* X starting in q_0 is $q_0, I_0, R_0, q_1, I_1, R_1, \ldots, q_m, I_m, R_m$. However, we require that the object's response at a state uniquely determines the new state, i.e. we assume if $\delta(q, I_i)$ contains (R_i, q_{i+1}) and (R_i, q'_{i+1}) then $q_{i+1} = q'_{i+1}$. Then we may denote the interval-sequential execution of X, starting in q_0 by $h = I_0, R_0, I_1, R_1, \ldots, I_m, R_m$, because the sequence of states $q_0, q_1, \ldots, q_m$ is uniquely determined by q_0, and by the sequences of inputs and responses. When we omit mentioning q_0 we assume there is some initial state in Q_0 that can produce h.

Note that X may be non-deterministic, in a given state q_i with input I_i it may move to more than one state and return more than one response. Sometimes it is convenient to require that the object is *total*, meaning that, for every singleton set $I \in 2^{Inv}$ and every state q in which the invocation inv in I is not pending, there is an $(R, q') \in \delta(q, I)$ in which there is a response to inv in R.

Our definition of interval-sequential execution is motivated by the fact that we are interested in *well-formed* executions $h = I_0, R_0, I_1, R_1, \ldots, I_m, R_m$. Informally, the processes should behave well, in the sense that a process does not invoke a new operation before its last invocation received a response. Also, the object should behave well, in the sense that it should not return a response to an operation that is not pending.

The *interval-sequential specification* of X, $ISSpec(X)$, is the set of all its interval-sequential executions.

Representation of interval-sequential executions In general, we will be thinking of an interval-sequential execution h as an alternating sequence of invoking and responding concurrency classes starting with an invoking class, $h = I_0, R_0, I_1, R_1, \ldots, I_m, R_m$. However, it is sometimes convenient to think of an execution as a a total order $\widehat{S} = (S, \xrightarrow{S})$ on a subset $S \subseteq CC(X)$, where $CC(X)$, is the set with all invoking and responding concurrency classes of X; namely, $h = I_0 \xrightarrow{S} R_0 \xrightarrow{S} I_1 \xrightarrow{S} R_1 \xrightarrow{S} \cdots \xrightarrow{S} I_m \xrightarrow{S} R_m$.

In addition, the execution $h = I_0, R_0, I_1, R_1, \ldots, I_m, R_m$ can be represented by a table, with a column for each element in the sequence h, and a row for

each process. A member $in \in I_j$ invoked by p_k (resp. a response $r \in R_j$ to p_k) is placed in the k'th row, at the $2j$-th column (resp. $2j+1$-th column). Thus, a transition of the automaton will correspond to two consecutive columns, I_j, R_j. See Figure 3, and several more examples in the figures below.

Remark 1. Let X be an interval-sequential object. Suppose for all states q and all I, if $\delta(q, I) = (R, q')$, then $|R| = |I|$, and additionally each $r \in R$ is a response to one $in \in I$. Then X is a *set-sequential* object. If in addition, $|I| = |R| = 1$, then X is a sequential object in the usual sense (see Figure 1).

3.3 An example: the validity task

Consider an object X with a single operation $\mathsf{validity}(x)$, that can be invoked by each process, with a *proposed* input parameter x, and a very simple specification: an operation returns a value that has been proposed. This problem is easily specified as a task. Indeed, many tasks include this property, such as consensus, set-agreement, write-snapshot, etc. As an interval-sequential object, it is formally specified by an automaton, where each state q is labeled with two values, $q.vals$ is the set of values that have been proposed so far, and $q.pend$ is the set of processes with pending invocations. The initial state q_0 has $q_0.vals = \emptyset$ and $q_0.pend = \emptyset$. If in is an invocation to the object, let $val(in)$ be the proposed value, and if r is a response from the object, let $val(r)$ be the responded value. For a set of invocations I (resp. responses R) $vals(I)$ denotes the proposed values in I (resp. $vals(R)$). The transition relation $\delta(q, I)$ contains all pairs (R, q') such that:

- If $r \in R$ then $id(r) \in q.pend$ or there is an $in \in I$ with $id(in) = id(r)$,
- If $r \in R$ then $val(r) \in q.vals$ or there is an $in \in I$ with $val(in) = val(r)$,
- $q'.vals = q.val \cup vals(I)$ and $q'.pend = (q.pend \cup ids(I)) \setminus ids(R)$.

On the right of Figure 3 there is part of a validity object automaton. On the left of Figure 3 it is illustrated an interval-sequential execution with the vertical red double-dot lines: I_0, R_0, I_1, R_1, where $I_0 = \{p.\mathsf{validity}(1), q.\mathsf{validity}(2)\}$, $R_0 = \{p.\mathsf{resp}(2)\}$, $I_1 = \{r.\mathsf{validity}(3)\}$, $R_1 = \{q.sfresp(3), r.\mathsf{resp}(1)\}$.

The interval-linearizability consistency notion described in Section 4 will formally define how a general execution (blue double-arrows in the figure) can be represented by an interval-sequential execution (red double-dot lines), and hence tell if it satisfies the validity object specification. Notice that the execution in Figure 3 shows that the validity object has no specification neither as a sequential nor as a set-sequential object.

4 Interval-Linearizability

Interval-sequential execution of the system Consider a subset $S \subseteq CC$ of the concurrency classes of the objects OBS in the system and an interval-sequential

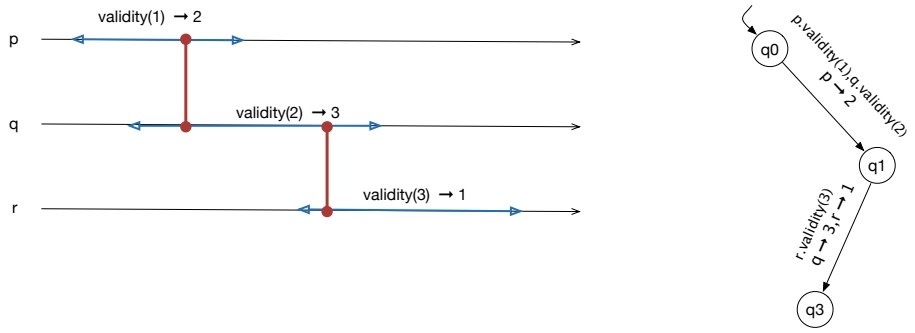


Fig. 3. An execution of a validity object, and the corresponding part of its interval-sequential automaton

execution $\widehat{S} = (S, \xrightarrow{S})$, defining an alternating sequence of invoking and responding concurrency classes, starting with an invoking class. For an object X, the *projection of* $\widehat{S}$ *at* X, $\widehat{S}|_X = (S_X, \xrightarrow{S_X})$, is defined as follows: (1) for every $C \in S$ with at least one invocation or response on X, S_X contains a concurrency class C', consisting of the (non-empty) subset of C of all invocations or responses of X, and (2) for every $C', C'' \in S_X$, $C' \xrightarrow{S_X} C''$ if and only if there are $T', T'' \in S$ such that $C' \subseteq T'$, $C'' \subseteq T''$ and $T' \xrightarrow{S} T''$.

We say that $\widehat{S} = (S, \xrightarrow{S})$ is an *interval-sequential execution of the system* if $\widehat{S}|_X$ is an interval-sequential execution of X for every $X \in OBS$. That is, if $\widehat{S}|_X \in ISSpec(X)$, the interval-sequential specification of X, for every $X \in OBS$. Let $\widehat{S} = (S, \xrightarrow{S})$ be an interval-sequential execution. For a process p, the *projection of* $\widehat{S}$ *at* p, $\widehat{S}|_p = (S_p, \xrightarrow{S_p})$, is defined as follows: (1) for every $C \in S$ with an invocation or response by p, S_p contains contains a class C with the invocation or response by p (there is at most one event by p in C), and (2) for every $a, b \in S_p$, $a \xrightarrow{S_p} b$ if and only if there are $T', T'' \in S$ such that $a \in T'$, $b \in T''$ and $T' \xrightarrow{S} T''$.

Interval-linearizability Recall that an execution of the system is a sequence of invocations and responses (Section 3.1). An invocation in an execution E is *pending* if it has no matching response, otherwise it is *complete*. An *extension* of an execution E is obtained by appending zero or more responses to pending invocations.

An *operation call* in E is a pair consisting of an invocation and its matching response. Let $comp(E)$ be the sequence obtained from E by removing its pending invocations. The order in which invocation and responses in E happened, induces the following partial order: $\widehat{OP} = (OP, \xrightarrow{op})$ where OP is the set with all operation calls in E, and for each pair $\mathsf{op}_1, \mathsf{op}_2 \in OP$, $\mathsf{op}_1 \xrightarrow{op} \mathsf{op}_2$ if and

only if $term(\mathsf{op}_1) < init(\mathsf{op}_2)$ in E, namely, the response of op_1 appears before the invocation of op_2. Given two operation op_1 and op_2, op_1 *precedes* op_2 if $\mathsf{op}_1 \xrightarrow{op} \mathsf{op}_2$, and they are *concurrent* if $\mathsf{op}_1 \overset{op}{\not\to} \mathsf{op}_2$ and $\mathsf{op}_2 \overset{op}{\not\to} \mathsf{op}_1$.

Consider an execution of the system E and its associated partial order $\widehat{OP} = (OP, \xrightarrow{op})$, and let $\widehat{S} = (S, \xrightarrow{S})$ be an interval-sequential execution. We say that an operation $a \in OP$ *appears* in a concurrency class $S' \in S$ if its invocation or response is in S'. Abusing notation, we write $a \in S'$. We say that $\xrightarrow{S}$ *respects* $\xrightarrow{op}$, also written as $\xrightarrow{op} \subseteq \xrightarrow{S}$, if for every $a, b \in OP$ such that $a \xrightarrow{op} b$, for every $T', T'' \in S$ with $a \in T'$ and $b \in T''$, it holds that $T' \xrightarrow{S} T''$.

Definition 1 (Interval-linearizability). *An execution E is* interval-linearizable *if there is an extension $\overline{E}$ of E and an interval-sequential execution $\widehat{S} = (S, \xrightarrow{S})$ such that*

1. *for every process p, $comp(\overline{E})|_p = \widehat{S}|_p$,*
2. *for every object X, $\widehat{S}|_X \in ISS(X)$ and*
3. *$\xrightarrow{S}$ respects $\xrightarrow{op}$, where $\widehat{OP} = (OP, \xrightarrow{op})$ is the partial order associated to $comp(\overline{E})$.*

We say that $\widehat{S} = (S, \xrightarrow{S})$ is an interval-linearization of E.

Remark 2. When we restrict to interval-sequential executions in which for every invocation there is a response to it in the very next concurrency class, then interval-linearizability boils down to set-linearizability. If in addition we demand that every concurrency class contains only one element, then we have linearizability. See Figure 1.

We can now complete the example of the validity object. In Figure 4 there is an interval linearization of the execution in Figure 3.

	init	*term*	*init*	*term*
p	validity(1)	$resp(2)$		
q	validity(2)			$resp(3)$
r			validity(3)	$resp(1)$

Fig. 4. An execution of a Validity object

Even though interval-linearizability is much more general than linearizability it retains some of its benefits.

Theorem 1 (Locality of interval-linearizability). *An execution E is interval-linearizable if and only if $E|_X$ is interval-linearizable, for every object X.*

Theorem 2 (Set-linearizability is non-blocking). *Let E be an interval-linearizable execution in which there is a pending invocation $inv(op)$ of a total operation. Then, there is a response $res(op)$ such that $E \cdot res(op)$ is interval-linearizable.*

5 Tasks and Interval-Sequential Objects

A task is a static way of specifying a *one-shot* concurrent problem, namely, a problem with one operation that can be invoked only once by each process. Here we study the relationship between tasks and the automata-based ways of specifying a concurrent problem that we have been considering.

A task is usually specified informally, in the style of Section 2. E.g., for the k-set agreement task one would say that each process proposes a value, and decides a value, such that (validity) a decided value has been proposed, and (agreement) at most k different values are decided.

Formally, a task $(\mathcal{I}, \mathcal{O}, \Delta)$ consists of an *input complex* $\mathcal{I}$, an *output complex* $\mathcal{O}$, and an input/output relation Δ. Each complex consists of a set of *simplexes*, of the form $s = \{(\mathrm{id}_1, x_1), \ldots, (\mathrm{id}_k, x_k)\}$, and closed under containment. An input (resp. output) simplex specifies an assignment of input (resp. output) values, x_i to process id_i. A singleton simplex is a *vertex*. The relation Δ specifies for each input simplex $s \in \mathcal{I}$, a sub-complex $\Delta(s) \subseteq \mathcal{O}$, such that if s, s' are two simplexes in $\mathcal{I}$ with $s' \subset s$, then $\Delta(s') \subset \Delta(s)$.

A task has only one operation, $\mathsf{task}()$, which process id_i may call with value x_i, if (id_i, x_i) is a vertex of $\mathcal{I}$. The operation $\mathsf{task}(x_i)$ may return y_i to the process, if (id_i, y_i) is a vertex of $\mathcal{O}$. Let E be an execution where each process calls $\mathsf{task}()$ once. Then, σ_E is the input simplex defined as follows: (id_i, x_i) is in σ_E iff in E there is an invocation of $\mathsf{task}(x_i)$ by process id_i. The output simplex τ_E is defined similarly: (id_i, y_i) is in τ_E iff there is a response y_i to a process id_i in E. We say that E *satisfies* $\langle I, O, \Delta \rangle$ if for every prefix E' of E, it holds that $\tau_{E'} \in \Delta(\sigma_{E'})$. The prefix requirement prevents executions that globally seem correct, but in a prefix a process predicts future invocations.[3]

From tasks to interval-sequential objects A task is a very compact way of specifying a distributed problem that is capable of describing allowed behaviors for certain concurrency patterns, and indeed it is hard to understand what exactly is the problem being specified. The following theorem (with its proof) provides an automata-based representation of a task, explaining which outputs may be produced in each execution, as permitted by Δ.

Theorem 3. *For every task T, there is an interval-sequential object O_T such that an execution E satisfies T if and only if it is interval-linearizable with respect to O_T.*

To give an intuition of this theorem, consider the immediate snapshot task for three processes in Figure 5 with two additional output simplexes, σ_1 and σ_2. A simple case is the output simplex in the center of the output complex, where the three processes output $\{p, q, r\}$. The case is simple because this simplex does not intersect the boundary, hence, it can be produced as output only when all three operations are concurrent and then the corresponding interval-sequential

[3] This requirement has been implicitly considered in the past by stating that an algorithm solves a task if any of its executions agree with the task specification.

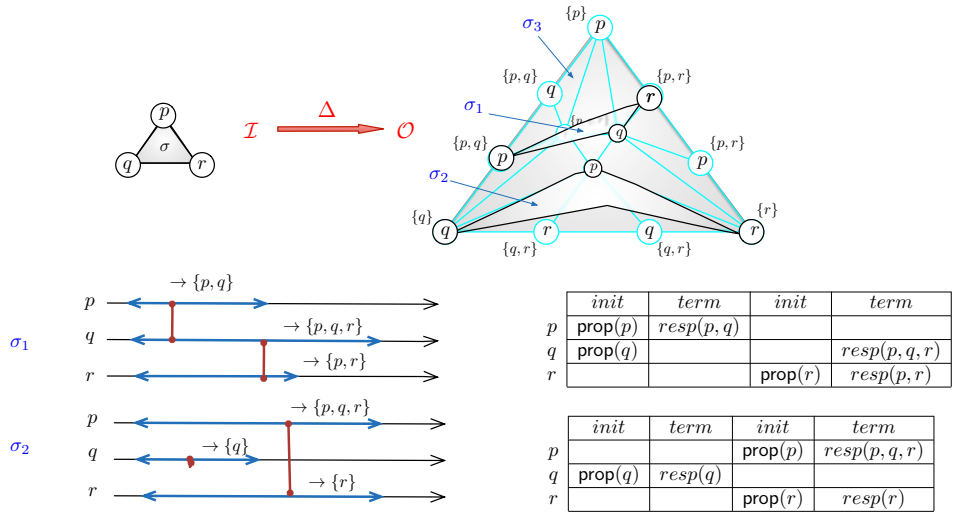


Fig. 5. .Two special output simplexes σ_1, σ_2, and interval-linearizations of two executions with corresponding outputs

object models this simplex with a single interval-sequential execution in which the three processes run concurrently. More interesting is the output simplex σ_3, where the processes also may run concurrently, but in addition, the same outputs may be returned in a fully sequential execution, because σ_3 intersects both the 0-dimensional (the corners) and the 1-dimensional boundary of the output complex. In fact σ_3 can also be produced if p, q are concurrent, and later comes r, because 2 vertices of σ_3 are in $\Delta(p, q)$ (such an execution is set-sequential).

Now, consider the two more awkward output simplexes σ_1, σ_2 in $\Delta(\sigma)$ added to the immediate-snapshot output complex, where $\sigma_1 = \{(p, \{p, q\}), (q, \{p, q, r\}), (r, \{p, r\})\}$, and $\sigma_2 = \{(p, \{p, q, r\}), (q, \{q\}), (r, \{r\})\}$. At the bottom of the figure, two executions and their interval-linearizations are shown, though there are more executions that are interval-linearizable and can produce σ_1 and σ_2. Consider σ_2. It has a face, $\{(q, \{q\})\}$, in $\Delta(\{q\})$, and another face, $\{(r, \{r\})\}$ in $\Delta(\{r\})$. This specifies a different behavior from the output simplex in the center, that does not intersect with the boundary. Since $\{(q, \{q\})\} \in \Delta(\{q\})$, it is OK for q to return $\{q\}$ when it invokes and returns before the others invoke. But also it is OK for q to return $\{q\}$ when it invokes and runs concurrently with p and r because $\{(q, \{q\})\} \in \Delta(\{p, q, r\})$. It similarly happens to r. Additionally since $\{(p, \{p, q, r\})\}$ is not in the boundary, p can return $\{p, q, r\}$ only if it runs concurrently with the others. The main observation here is that the structure of the mapping Δ encodes the interval-sequential executions that can produce the outputs in a given output simplex. In the example, Δ precludes the possibility

that in a sequential execution the processes outputs the values in σ_1, since Δ specifies no process can decide without seeing anyone else.

From one-shot interval-sequential objects to tasks The converse of Theorem 3 is not true. Lemma 1 shows that even some sequential objects, such as one-shot queues, cannot be represented as a task. Also, recall that there are tasks with no set-sequential specification. Thus, both tasks and set-sequential objects are interval-sequential objects, but they are incomparable.

Lemma 1. *There is a sequential one-shot object O such that there is no task T_O, satisfying that an execution E is linearizable with respect to O if and only if E satisfies T_O (for every E).*

While this version of tasks have strictly less expressive power than interval-sequential one-shot objects, a slightly different version has the same power for specifying distributed one-shot problems. Roughly, tasks cannot model interval-sequential objects because they do not have a mechanism to encode the state of an object. The extension below allows to model states.

In a *refined* task $T = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$, $\mathcal{I}$ is defined as usual and each output vertex of $\mathcal{O}$ has the form $(id_i.y_i, \sigma'_i)$ where id_i and y_i are, as usual, the ID of a process and an output value, and σ'_i is an input simplex called the *set-view* of id_i. The properties of Δ are maintained and in addition it satisfies the following: for every $\sigma \in \mathcal{I}$, for every $(id_i, y_i, \sigma'_i) \in \Delta(\sigma)$, it holds that $\sigma'_i \subseteq \sigma$. An execution E *satisfies* a refined task T if for every prefix E' of E, it holds that $\Delta(\sigma_{E'})$ contains the simplex $\{(id_i, y_i, \sigma_{i\,E''}) : (id_i, y_i) \in \tau_{E'} \wedge E''$ (which defines $\sigma_{i\,E''}$) is the shortest prefix of E' containing $(id_i, y_i)\}$.

We stress that, for each input simplex σ, for each output vertex $(id_i, y_i, \sigma_i) \in \Delta(\sigma)$, σ_i is a way to model distinct output vertexes in $\Delta(\sigma)$ whose output values (in (id_i, y_i)) are the same, then a process that outputs that vertex does not actually output σ_i. In fact, the set-view of a process id_i corresponds to the set of invocations that precede the response (id_i, y_i) to its invocation in a given execution (intuitively, the invocations that a process "sees" while computing its output value). Set-views are the tool to encode the state of an object. Also observe that if E satisfies a refined task T, then the set-views behave like snapshots: (1) a process itself (formally, its invocation) appears in its set-view and (2) all set-view are ordered by containment (since we assume E is well-formed).

Theorem 4. *For every one-shot interval-sequential object O with a single total operation, there is a refined task T_O such that any execution E is interval-linearizable with respect to O if and only if E satisfies T_O.*

Theorem 5. *For every refined task T, there is an interval-sequential object O_T such that an execution E satisfies T if and only if it is interval-linearizable with respect to O_T.*

References

1. Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890 (1993)

2. Afek Y., Gafni E., and Lieber O., Tight group renaming on groups of size g is equivalent to g-consensus. DISC 2009. Springer LNCS 5805, pp. 111–126 (2009).
3. Aspnes J. and Ellen F., Tight bounds for adopt-commit objects. *Theory Computing Systems*, 55(3): 451–474 (2014).
4. Borowsky E. and Gafni E., Immediate atomic snapshots and fast renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, ACM Press, pp. 41–51 (1993)
5. Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG distributed simulation algorithm. *Distributed Computing* 14(3): 127–146 (2001)
6. Chandra T.D., Hadzilacos V., Jayanti P., Toueg S., Generalized irreducibility of consensus and the equivalence of t-resilient and wait-free implementations of consensus. *SIAM Journal of Computing* 34(2): 333–357 (2004)
7. Castañeda A., Rajsbaum S., and Raynal M., Specifying Concurrent Problems: Beyond Linearizability. `http://arxiv.org/abs/1507.00073`
8. Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158 (1993)
9. Conde R., Rajsbaum S., The complexity gap between consensus and safe-consensus. *21th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'14)*, Springer LNCS 8576, pp. 68–82 (2014)
10. Filipović I., O'Hearn P., Rinetky N., and Yang H., Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51–52):4379–4398 (2010)
11. Friedman R., Vitenberg R., and Chokler G., On the composability of consistency conditions. *Information Processing Letters*, 86(4):169–176, 2003.
12. Gafni E., Round-by-round fault detectors: unifying synchrony and asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 143–152 (1998)
13. Gafni E., Snapshot for time: the one-shot case. *arXiv:1408.3432v1*, 10 pages, (2014)
14. van Glabbeek R.J., On the expressiveness of higher dimensional automata. *Theoretical Computer Science*, 356(3):265–290 (2006)
15. Gotsman A., Musuvathi M. and Yang H., Show no weakness: sequentially consistent specifications of TSO libraries. *DISC*. Springer LNCS 7611, pp. 31–45 (2012)
16. Hemed N. and Rinetzky N., Brief Announcement: Concurrency-aware linearizability. *Proc. 33th ACM Symposium on Principles of Distributed Computing (PODC'14)*, page 209–211, ACM Press (2014). Full version to appear in these proceedings.
17. Herlihy M., Kozlov D., and Rajsbaum S., *Distributed computing through combinatorial topology*, Morgan Kaufmann (2014)
18. Herlihy M., Rajsbaum S., Raynal M., Power and limits of distributed computing shared memory models. *Theoretical Computer Science*, 509: 3–24 (2013)
19. Herlihy M. and Shavit N., *The art of multiprocessor programming.* Morgan Kaufmann (2008)
20. Herlihy M. and Wing J., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Progr. Lang. and Systems*, 12(3):463–492 (1990)
21. Neiger G., Set-linearizability. Brief announcement in *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, ACM Press, page 396 (1994)
22. Raynal M., *Concurrent programming: algorithms, principles, and foundations.* Springer (2013)
23. Scherer III W. and Scott M. L., Nonblocking Concurrent Data Structures with Condition Synchronization. *Proc. DISC*. Springer LNCS 3274, pp. 174–187, (2004)
24. Shavit N., Data structures in the multicore age. *Comm. ACM,* 54(3):76–84 (2011)